Using on-the-fly Verification Techniques for the Generation of Test Suites

Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, César Viho

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Using on-the-fly verification techniques for the generation of test suites

Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, César Viho

N ° 2987

16 Septembre 1996

THÈME 1

ISSN 0249-6399

INRIA
RHÔNE-ALPES

Using on-the-fly verification techniques for the generation of test suites

Jean-Claude Fernandez, Claude Jard, Thierry Jéron, Laurence Nedelka, César Viho

Thème 1 — Réseaux et systèmes
Projets Pampa et Spectre

Rapport de recherche n°2987 — 16 Septembre 1996 — 12 pages

Abstract: In this paper we attempt to demonstrate that on-the-fly techniques, developed in the context of verification, can help in deriving test suites. Test purposes are used in practice to select test cases according to some properties of the specification. We define a consistency preorder linking test purposes and specifications. We give a set of rules to check this consistency and to derive a complete test case with preamble, postamble, verdicts and timers. The algorithm, which implements the construction rules, is based on a depth first traversal of a synchronous product between the test purpose and the specification. We shortly relate our experience on an industrial protocol with TGV, a first prototype of the algorithm implemented as a component of the CADP toolbox.

Key-words: generation of test suite, on the fly verification, test purpose, formal specification, inputs–outputs transitions systèmes, verdicts, timers

(Résumé : tsvp)

This work has been partially supported by an industrial contract with Verilog in a study for the french DGA (Direction Générale pour l'Armement)

This paper was published in CAV'96, LNCS 1102, Springer Verlag

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : (33) 76 61 52 00 – Télécopie : (33) 76 61 52 52

Génération automatique de séquences de test par la technologie de vérification

Résumé : Dans ce rapport nous montrons que les techniques de vérification à la volée peuvent être appliquées à la génération de séquences de test. Les objectifs de test sont utilisés, en pratique, pour sélectionner des séquences de test. Nous définissons une relation de cohérence entre la spécification et l'objectif de test. Nous donnons un ensemble de règles pour construire une séquence de test complète (préambule, corps de test, postambule), pour définir les verdicts et les timers. L'algorithme, qui implémente les règles de construction, parcourt en profondeur d'abord, un produit synchrone entre la spécification et l'objectif de test. Cet algorithme est linéaire. Nous décrivons brièvement notre expérience sur un protocol en utilisant un prototype appelé TGV. Ce prototype a été développé dans la boite à outils CADP.

Mots-clé : génération de séquences de test, vérification à la volée, objectif de test, spécification formelle, système de transitions à entrées–sorties, verdicts, timers.

1 Introduction

It is widely recognized that testing is an essential component of the full lifecycle of communicating systems. However, the process of generating test suites is complicated, error-prone and expensive. The intrinsic difficulty comes from the black-box nature of the implementation: its behaviour is only observable and controlable at the interfaces. In that context, a formal framework is a prerequisite for giving precise and consistent meanings of test verdicts. The usual theoretical approach [2] is to consider a formal specification of the intended behaviour of the Implementation Under Test (IUT). It permits to define the notion of *conformance* relation linking an implementation to the specification and the notion of *verdict* associated to the application of a test case (set of interaction sequences) to an implementation, w.r.t. the conformance relation. The problem is to automatically generate correct test cases from a formal specification of the IUT. A correct test case, applied to an IUT, will declare "fail" only implementations which do not conform to the specification (*soundness* property). We also require that an implementation which does not conform to the specification might be detected by repeating the application of a test case, under a fairness assumption on the implementation (property of exhaustivity).

During the last decade, testing theory and algorithms for the generation of tests have been developed from Labelled Transition System specifications (LTS). Test generation involves sub-problems of traversal, comparison or reduction of LTS, already addressed by verification. Consequently, we think, among others [4], that time is ripe for linking test and verification. The experience of practitioners tells them that it is not reasonable to try to validate all possible behaviours of their protocol. It is why they use informal test purposes. Basically, we traverse in a depth-first manner, the synchronous product of the IUT specification and of the test purpose. During the traversal, we check their mutual consistency. If so, an acyclic test graph is generated and decorated with verdicts and timers. The algorithm is an original extension of the on-the-fly verification kernel we developed a few years ago [11, 12, 8, 9]. It provides a complete treatment of the problem of test cases, including preambles and postambles, verdicts and timers management. It is now well known that depth-first traversals are the heart of some good verification algorithms, for behavioural comparison and reduction [8], as well as for model-checking [11, 5, 12]. We show that it is also true for test generation, which constitutes a good example of transfer from verification to testing.

The test generator, based on verification technology, has been prototyped in the context of an industrial consortium, linking Vérilog, Cap-Sesa, Cnet, Inria and the French Army. An experiment was performed on a real ISDN protocol specification. The results were very encouraging, confirming the interest of using this kind of algorithmic, which is now mature enough to be transfered in the industrial world, to deal with real formal specifications. Our approach is compatible with symbolic (or structural) ones like TVEDA [14] which may compute test purposes using reachability analysis.

The presentation is organized as follows. We start by defining the different models used for describing test purposes, test cases, the specification and the IUT. We define a consistency preorder between test purposes and specifications, and a test conformance relation linking implementations to specifications. We give the formal rules allowing the construction of a test case from a test purpose and a specification. We give some results concerning soundness and exhaustivity of our generated test cases. Finally, we give the main results gained during an experiment on an ISDN protocol.

2 Models

In this section we first describe the models used for the description of the different objects involved in the generation of test cases. They are used to define the notion of consistency relating a test purpose with a specification and the notion of conformance relating an implementation with a specification. These models are then used to define formal rules for the construction of test cases.

2.1 Input-Outputs labelled transition systems

The models used are all based on Input-Output Labelled Transition Systems (IOLTS) in which input and output actions are differentiated because of the asymmetrical nature of the testing activity.

We consider a finite alphabet of actions A, partitioned into two sets: *input actions* A_I and *output actions* A_O. We shall let α, β range over A, i, i' range over A_I and o, o' range over A_O. We consider finite IOLTS $\mathrm{M}=(Q^{\mathrm{M}}, A, T^{\mathrm{M}}, q^{\mathrm{M}}_{\mathrm{init}})$ where Q^{M} is the set of states, $q^{\mathrm{M}}_{\mathrm{init}}$ is the initial state, $T^{\mathrm{M}} \subseteq Q^{\mathrm{M}} \times A \times Q^{\mathrm{M}}$ is the transition relation.

We adopt the following notations and conventions: Let $\sigma \in A^*, p, q \in Q^{\mathrm{M}}$. We write $p \xrightarrow{\alpha}_{\mathrm{M}} q$ iff $(p, \alpha, q) \in T^{\mathrm{M}}$ and write $p \stackrel{\sigma}{\Longrightarrow}_{\mathrm{M}} q$ iff $\exists \alpha_1, \alpha_2 \cdots \alpha_n \in A$, $p_0, \cdots, p_n \in Q^{\mathrm{M}}$. $\sigma = \alpha_1.\alpha_2 \ldots \alpha_n$ and $p_0 = p$, $p_i \stackrel{\alpha_{i+1}}{\rightarrow}_{\mathrm{M}} p_{i+1}$ with $i < n$, $p_n = q$. $\mathcal{A}(q) = \{\alpha \mid \exists q' \text{ and } q \xrightarrow{\alpha}_{\mathrm{M}} q'\}$ is the set of immediate actions after q, $\mathcal{I}(q) = \mathcal{A}(q) \cap A_I$ is the set of inputs after q, and $\mathcal{O}(q) = \mathcal{A}(q) \cap A_O$ is the set of outputs after q. $\mathrm{Succ}_\alpha(q) = \{q' \mid q \xrightarrow{\alpha}_{\mathrm{M}} q'\}$ is the set of states reachable from q by means of a transition labelled by α. We write $\neg(p \xrightarrow{\alpha}_{\mathrm{M}})$ if there is no transition starting from p and labelled by α, $\neg(p \xrightarrow{\alpha}_{\mathrm{M}}) = (\mathrm{Succ}_\alpha(p) = \emptyset)$. We note p **after** $\sigma = \{q \in Q^{\mathrm{M}} \mid p \stackrel{\sigma}{\Longrightarrow}_{\mathrm{M}} q\}$ the set of states reachable from p by the sequence of transitions σ and $traces(p) = \{\sigma \in A^* | p \textbf{ after } \sigma \neq \emptyset\}$ the set of sequences starting from p and reaching a state in Q^{M}. In the sequel, we will not distinguish between a transition system and its initial state.

An IOLTS satisfies the *controlability condition* if and only if for each state, if an output is enabled, then there is exactly one outgoing transition. More formally, if $|X|$ denotes the cardinality of the set X, $\forall p . |\mathcal{O}(p)| = 0 \vee (|\mathcal{O}(p)| = 1 \wedge \mathcal{O}(p) = \mathcal{A}(p))$.

An IOLTS is *deterministic* if and only if $\forall p, \forall \sigma \,.\, |p \textbf{ after } \sigma| \leq 1$.

We consider four kinds of IOLTS: the specification, the implementation, the test purpose behaviour and the test cases which meanings are described below.

2.2 Specification and implementation

An IUT is placed in a test environment in which the tester can only interact with inputs and outputs. Thus the tester has an *external view* of the implementation. In contrast, a specification generally models the *internal view* of the system, i.e. the behaviour of the system with its internal actions, without considering the way it interacts with the environment. But this interaction should be taken into account in the test generation. As an example, if the implementation communicates asynchronously with its environment through several points of control and observation (PCOs), two subsequent and causally ordered outputs may be observed by the environment as two concurrent inputs if they occur on two different PCOs. In the following, we will consider the environmental point of view: outputs are *controlable* actions initiated by the environment (which may be the tester) and sent to the IUT whereas inputs are *observable* actions, initiated by the IUT and received by the environment.

While testing an IUT, we check for the conformance of the IUT in its environment with the specification in the same environment. Thus we first have to transform the specification into its external view. Internal actions which are not observable by the environment have to be hidden and replaced by a τ transition. Inputs are replaced by outputs and vice versa, taking care of concurrency which may be produced by asynchronous interaction. This is called the mirror image operation. After that, we have to apply a τ-reduction which suppresses τ transitions. The transition system of the resulting specification is then an IOLTS. This has been implemented on-the-fly in our prototype and we will not give more details of how this can be done effectively. Deadlocks are often supposed to be observable by a tester. In practice the tester uses timers to achieve this (see 3.2.2) and we have to suppose that a timeout occurs if and only if the implementation is deadlocked. This is why timeouts are considered as inputs of the tester. If the specification is allowed to deadlock in a particular state, this is modelled by a special transition δ considered as an input of the environment initiated by the system. This treatment of deadlocks is quite similar with what is done in [15, 13]. Finally, the last operation is determinization.

The resulting specification is a deterministic IOLTS $S = (Q^{\mathrm{S}}, A, T^{\mathrm{S}}, q^{\mathrm{S}}_{\mathrm{init}})$ with $A = A_I \cup A_O$ and $\delta \in A_I$ a distinguished input. Without loss of generality, we will suppose that S starts with outputs of the environment i.e. $\mathcal{A}(q^{\mathrm{S}}_{\mathrm{init}}) \subseteq A_O$. In the following, specification will always correspond to the external view S of the specification. Though the implementation is not necessarily a transition system (it may be a physical system), as in all testing theories, we have to reason formally about it and model its behaviour. As it is only considered by its interactions with the environment, it is also modelled as an IOLTS $I = (Q^{\mathrm{I}}, A^{\mathrm{I}}, T^{\mathrm{I}}, q^{\mathrm{I}}_{\mathrm{init}})$, with $A^{\mathrm{I}} = A^{\mathrm{I}}_I \cup A^{\mathrm{I}}_O$, $A_I \subseteq A^{\mathrm{I}}_I$.

2.3 Test Purpose

A test purpose defines a property on some particular interactions between the IUT and the tester. It consists in two parts : a behavioural part and a constraint part. The constraint part gives some property on the state of the implementation. It can be seen as computable by the environment and will be modelled by an input for the tester. Thus it is integrated in the behavioural part.

Definition 2.1 *(Test Purpose behaviour) A test purpose behaviour is a deterministic acyclic IOLTS* $TP = (Q^{TP}, A, T^{TP}, q^{TP}_{init})$ *satisfying the controlability condition and with a set of distinguished states* `Accept`$\subseteq Q^{TP}$ *with no successor.*

2.4 Test Cases

A *test case* is a set of sequences of actions describing all the interactions occuring between an IUT and a tester which wants to verify that an implementation conforms with the specification according to a test purpose. In an industrial context, test cases are often described using the Tree and Tabular Combined Notation (TTCN [10]). Some transitions are decorated with verdicts with the following informal meaning :
(PASS): means that the test purpose is satisfied by the current sequence. But a sequence leading to the initial state (Postamble) must be applied in order to carry on another test case. It is a temporary verdict as the application of the postamble may produce Fail verdicts.
PASS: this is a definitive verdict meaning that the initial state has been reached after a (PASS) verdict. The sequence between (PASS) and PASS is a Postamble.
FAIL: means non-conformance of the IUT.
INCONCLUSIVE: this verdict is used in practice when a reception is allowed in the specification but cannot lead to a (PASS) or leads to a behaviour that is not considered in the test case because testing cannot be exhaustive in practice.

Definition 2.2 *(Test Case) A test case is a deterministic acyclic IOLTS* $TC = (Q^{TC}, A, T^{TC}, q^{TC}_{init})$ *satisfying the controlability condition. A test suite is a set of test cases.*

2.5 Consistency and test conformance relation

In this section, we define what we mean by consistency of a test purpose w.r.t a specification and which conformance relation linking the implementation with its specification is considered.

A test purpose TP is said to be *consistent* w.r.t a specification S, denoted by $q^{\mathrm{TP}}_{\mathrm{init}} \prec q^{\mathrm{S}}_{\mathrm{init}}$, if the two following conditions are satisfied: the set of behaviours described by the test purpose is included (see the definition below) in the set of behaviours of the specification, and from each state of S corresponding to an `Accept` state of TP, there is a path in S to $q^{\mathrm{S}}_{\mathrm{init}}$.

Definition 2.3 *(Consistency preorder). A relation* $R \subseteq Q^{TP} \times Q^{S}$ *is a* consistency relation *if and only if* $R \subseteq \mathcal{F}(R)$ *where,*

$$
\begin{array}{lcl}
\mathcal{F}(R) = \{(p^{TP}, p^{S}) \mid & & \\
(\forall \alpha, \forall q^{TP} \cdot p^{TP} \stackrel{\alpha}{\rightarrow}_{TP} q^{TP} & \Longrightarrow & \exists q^{S}, q^{S}_{1}, \exists \sigma \in (A \setminus \{\alpha\})^{*} \cdot p^{S} \stackrel{\sigma}{\Longrightarrow}_{S} q^{S}_{1} \stackrel{\alpha}{\rightarrow}_{S} q^{S} \wedge \\
 & & (q^{TP}, q^{S}) \in R \wedge (p^{TP}, q^{S}_{1}) \in R) \wedge \\
p^{TP} \in \texttt{Accept} & \Longrightarrow & \exists \sigma \in A^{*} \cdot p^{S} \stackrel{\sigma}{\Longrightarrow}_{S} q^{S}_{init}\}
\end{array}
$$

$q^{TP}_{init} \prec q^{S}_{init}$ *if and only if there is a relation* $R \subseteq \mathcal{F}(R)$*, containing* $(q^{TP}_{init}, q^{S}_{init})$

If the test purpose and the specification are consistent, we can derive sound test cases. A test case is sound if it gives a negative verdict only if the implementation is not correct w.r.t. the specification.

We consider a conformance relation quite similar to those in [15, 13]. Informally, the conformance relation states that outputs of the environment which are not accepted by the specification may be accepted by the implementation but inputs produced by the implementation must be also produced by the specification.

Definition 2.4 *(Test conformance relation) Let* S *and* I *be two IOLTS describing the external view of a specification and an implementation,*

$$I \textbf{ ioconf } S \textit{ if and only if } \forall \sigma \in traces(S), \mathcal{I}(I \textbf{ after } \sigma) \subseteq \mathcal{I}(S \textbf{ after } \sigma)$$

3 Construction rules

The essence of the on-the-fly method is to traverse a kind of synchronous product between two graphs, one for the specification and the other for the property to be checked. We first define this synchronous product. Then we give the rules for the test case construction, including decoration with verdicts and timers. Finally we give some properties of the generated test cases.

3.1 Synchronous product

A transition is firable in the product if either it is firable in the two components or it is firable only in the specification.

Definition 3.1 *(synchronous product) We define the product* $P = (Q^P, A, T^P, (q^{TP}_{init}, q^S_{init}))$, *with* $Q^P \subseteq Q^{TP} \times Q^S$ *where* Q^P *and* T^P *are the smallest sets obtained by application of the following rules:*

- `[Sync1]` $(q^{TP}_{init}, q^S_{init}) \in Q^P$,
- `[Sync2]` $$\frac{(p^{TP}, p^S) \in Q^P \quad p^{TP} \xrightarrow{\alpha}_{TP} q^{TP} \quad p^S \xrightarrow{\alpha}_S q^S}{(q^{TP}, q^S) \in Q^P \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}$$
- `[Sync3]` $$\frac{(p^{TP}, p^S) \in Q^P \quad \neg(p^{TP} \xrightarrow{\alpha}_{TP}) \quad p^S \xrightarrow{\alpha}_S q^S \quad (p^{TP}, p^S) \notin \mathbf{Accept} \times \{q^S_{init}\}}{(p^{TP}, q^S) \in Q^P \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (p^{TP}, q^S)}$$

3.2 The test case construction

The algorithm is based on a depth first traversal of the synchronous product, definition 3.1. Two mains actions are performed : the consistency relation between the test purpose and the specification is checked while a *direct acyclic graph* DAG is synthesized, definition 3.2.1. More precisely, a stack stores the states of the current execution sequence. The algorithm proceeds as follows, starting from the initial state, $(q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}})$. Let $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ be the current state, i.e. the top of the stack, and $(p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}})$ a transition not yet analyzed. If q^{TP} is an accepting state, then a postamble is computed by searching a shortest path from q^{S} to $q^{\mathrm{S}}_{\mathrm{init}}$; else if $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ belongs to the DAG then the transition is added to the DAG; otherwise, if $(q^{\mathrm{TP}}, q^{\mathrm{S}})$ does not belong to the stack nor to the visited states, then the state $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ is pushed on the stack. When all the transitions starting from the state $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ are analyzed, then $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ is popped in the set of visited states. The operator Comb is used in order to ensure the controlability condition. The algorithm terminates when the stack is empty and succeeds if $(q^{\mathrm{TP}}_{\mathrm{init}}, q^{\mathrm{S}}_{\mathrm{init}})$ belongs to the preamble and if `Accept` $\times \{q^{\mathrm{S}}_{\mathrm{init}}\}$ is a subset of the visited states. The algorithm requires a time complexity linear with respect to the size of the transition relation of the synchronous product and a space complexity linear with respect to its state space.

Definition 3.2 *We define DAGs syntactically as* $n ::= 0 \mid 1 \mid \alpha; n \mid n + n$

We also define a predicate $\vdash^v$ *for* $v = 1, 2, 3$. $\vdash^v (p^{TP}, p^S) : n$ *means "the node associated with* (p^{TP}, p^S) *is* n *and belongs to the preamble, test case body and postamble if* v *is respectively* 1, 2 *or* 3*". We use* $Node((p^{TP}, p^S))$ *to denote the node currently associated to* (p^{TP}, p^S). *Initially* $Node((p^{TP}, p^S)) = 0$. *An operator Comb is used to accumulate the nodes in order to ensure the controlability condition:*

$$Comb(m, \alpha; n) := \begin{cases} \alpha; n \text{ if } \alpha \in A_O,\ m = \Sigma \alpha_i; n_i \text{ and } \forall i, \alpha_i \in A_I \\ m \text{ if } m = \alpha'; n' \text{ and } \alpha' \in A_O \\ m + \alpha; n \text{ otherwise} \end{cases}$$

Preamble

$$\frac{v \in \{1,2\} \quad \vdash^v (q^{TP}_{init}, q^S) : n \quad (q^{TP}_{init}, p^S) \xrightarrow{\alpha}_P (q^{TP}_{init}, q^S)}{\vdash^1 (q^{TP}_{init}, p^S) : Comb(Node((q^{TP}_{init}, p^S)), \alpha; n)}$$

Test Case Body

$$\frac{\vdash^2 (q^{TP}, q^S) : n \quad q^{TP} \neq q^{TP}_{init} \quad (q^{TP}_{init}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}{\vdash^2 (q^{TP}_{init}, p^S) : Comb(Node((q^{TP}_{init}, p^S)), \alpha; n)}$$

$$\frac{\vdash^2 (q^{TP}, q^S) : n \quad p^{TP} \neq q^{TP}_{init} \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}{\vdash^2 (p^{TP}, p^S) : Comb(Node((p^{TP}, p^S)), \alpha; n)}$$

$$\frac{\vdash^3 (q^{TP}, q^S) : n \quad p^{TP} \notin \mathtt{Accept} \quad q^{TP} \in \mathtt{Accept} \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (q^{TP}, q^S)}{\vdash^2 (p^{TP}, p^S) : Comb(Node((p^{TP}, p^S)), \alpha; n)}$$

Postamble

$$\vdash^3 (p^{TP}, q^S_{init}) : 1 \quad (p^{TP} \in \mathtt{Accept})$$

$$\frac{\vdash^3 (p^{TP}, q^S) : n \quad (p^{TP}, p^S) \xrightarrow{\alpha}_P (p^{TP}, q^S) \quad p^{TP} \in \mathtt{Accept}}{\vdash^3 (p^{TP}, p^S) : Comb(Node((p^{TP}, p^S)), \alpha; n)}$$

3.2.1 The test case verdicts

We define the partial function `verdict` which assigns verdicts to some transitions in the DAG which construction is defined above. This function is defined by means of the rules below.

We complete the definition of the predicate $\vdash^v$ for $v = 4, 5$. $\vdash^4 (p^{\mathrm{TP}}, p^{\mathrm{S}}) : 1$ means that "the node associated with $(p^{\mathrm{TP}}, p^{\mathrm{S}})$ is 1 and is the ending state of a transition labelled by an Inconclusive. We need a new state *Fail_State* in the synchronous product and the axiom $\vdash^5 Fail\_State : 1$.

In the sequel, u, v range over $1, 2, 3, 4, 5$.

Pass : assigned to a transition in the DAG if the ending state is in the Postamble and the specification is in state $q^{\mathrm{S}}_{\mathrm{init}}$

$$\frac{\vdash^v (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n \quad v \in \{2,3\} \quad \vdash^3 (q^{\mathrm{TP}}, q^{\mathrm{S}}_{\mathrm{init}}) : 1 \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}}_{\mathrm{init}})}{\mathtt{verdict}(n, \alpha, 1) = \mathrm{Pass}}$$

(Pass) : assigned to a transition linking a state of the Test Case Body to a state in the Postamble.

$$\frac{\vdash^2 (p^{\mathrm{TP}}, p^{\mathrm{S}}) : m \quad \vdash^3 (q^{\mathrm{TP}}, q^{\mathrm{S}}) : n \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}})}{\mathtt{verdict}(m, \alpha, n) = (\mathrm{Pass})}$$

Inconclusive : add a new transition with verdict Inconclusive from a state in the DAG which allows an input reaching a state not in the DAG

$$\frac{\vdash^v (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n \quad \not\vdash^u (q^{\mathrm{TP}}, q^{\mathrm{S}}) \quad u, v \in \{1,2,3\} \quad (p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{i}_{\mathrm{P}} (q^{\mathrm{TP}}, q^{\mathrm{S}}) \quad i \in A_I}{\vdash^4 (q^{\mathrm{TP}}, q^{\mathrm{S}}) : 1 \quad (n, i, 1) \in DAG \quad \mathtt{verdict}(n, i, 1) = \mathrm{Inconclusive}}$$

Fail : in each state of the DAG, an input of the implementation which is not allowed in the specification should produce a Fail verdict. A new transition with verdict Fail is (virtually) added. In practice this corresponds to an *Otherwise Fail* in TTCN.

$$\frac{\vdash^v (p^{\mathrm{TP}}, p^{\mathrm{S}}) : n \quad v \in \{1,2,3\} \quad ((i \in A_I, \neg((p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{i}_{\mathrm{P}})) \text{ or } i \in A^{\mathrm{I}}_I \setminus A_I)}{\vdash^5 Fail\_State : 1 \quad (n, i, 1) \in DAG \quad \mathtt{verdict}(n, i, 1) = \mathrm{Fail}}$$

3.2.2 Timers

Timers are useful in practice in order to insure against implementation deadlocks. The management of timers is made on the DAG generated by the test generation rules 3.2. As timers depend on inputs, we associate a timer t_i to each input i labelling a transition in the test case. Three operations on a timer are available: $\mathtt{Start}(t_i)$ which initializes the timer and must be done as soon as input i is expected, $\mathtt{Cancel}(t_i)$ which is done when i is received or when, due to a choice, i is no more expected, and $\mathtt{Timeout}(t_i)$ which represents the observation of a deadlock when waiting for i.

Let $(p^{\mathrm{TP}}, p^{\mathrm{S}}) \xrightarrow{\alpha}_{\mathrm{T}} (q^{\mathrm{TP}}, q^{\mathrm{S}})$ be a transition of the synchronous product and $t : (n, \alpha, m)$ the corresponding transition in the DAG. Let $\mathtt{Ind}((p^{\mathrm{TP}}, p^{\mathrm{S}}))$ be the independency relation which represents the concurrency. The independency relation is a binary symmetrical relation defined on the inputs of a state: two inputs are independant if they may be received in any order. We denote by $Running(n)$ the set of timers that have been started in the sequences leading to n and have not yet been cancelled, Cl and St are sets of timers that have to be respectively cancelled or started after action α. Finally, $\mathtt{discard}(t)$ means that t is discarded from the DAG. The following rules specify the timers management:

Init As specifications, test cases start with an output, thus if r is the root of the DAG, $Running(r) = \emptyset$

Cancel and Start

$$\frac{t : (n, \alpha, m) \in DAG \quad Running(n) = R}{\begin{array}{c} t' : (n, \alpha; \mathtt{Cancel}(Cl); \mathtt{Start}(St), m) \in DAG \quad Running(m) = (R \setminus Cl) \cup St \\ \mathtt{discard}(t) \end{array}}$$

where

$$Cl = \{t_i | i \in \mathcal{I}((p^{\mathrm{TP}}, p^{\mathrm{S}})) \wedge (\alpha, i) \notin \mathtt{Ind}((p^{\mathrm{TP}}, p^{\mathrm{S}}))\}$$
$$St = \{t_i | i \in \mathcal{I}((p^{\mathrm{TP}}, p^{\mathrm{S}}))\} \setminus (R \setminus Cl)$$

i.e. all timers corresponding to inputs not concurrent with α must be cancelled and a timer must be started for each input available in m if it is not already running in n, except if it has just been cancelled.

Timeouts We suppose that $\delta \in A_I$. By the construction and verdict rules, in each node of the DAG, there is a transition labelled δ and its verdict may be (PASS) if δ is in the test purpose, Inconclusive if it is in the specification or Fail otherwise. If an input i (i may be δ) is possible in a state of the synchronous product, a transition labelled by $\mathtt{Timeout}(t_i)$ is added. The verdict assigned to this timeout must be the same as the verdict assigned to δ.

$$\frac{t : (n, i, m) \in DAG \quad i \in A_I \quad \mathtt{verdict}(t) \neq Fail \quad t' : (n, \delta, 1) \in DAG}{t'' : (n, \mathtt{Timeout}(t_i), 1) \in DAG \quad \mathtt{verdict}(t'') = \mathtt{verdict}(t')}$$

Discard δ For each transition $t : (n, \delta, 1) \in DAG$, apply $\texttt{discard}(t)$

Another depth first search is performed on the DAG to generate the timers operations. Unlike the DAG construction, which works by synthesis (just around the pop operation) the operations on timers are generated before the exploration of the state successors (around the push operation). The running set associated with each state is initialized to empty set at the initial state. It is inherited from a state to its successor. During this step, on one hand, each transition of the DAG is decorated with **cancel** and **start** operations on timers, on the other hand some transitions labelled by **timeout** are added, following the previous rules.

3.3 Exemple

An example of a synchronous product with the associated dag is given figure 1. Loops are discarded and the depth-first traversal stops when the initial state of the specification is reached while an accepting state of the test purpose is reached. An example of such a state is $(2, 0)$. Inputs are replaced by outputs and vice versa. The dag satisfies the *controlability* condition which says that the tester controls its ouputs.

The transition of the dag are decorated with verdicts. A timer **tm** is associated with each input **tm**, see figure 2.

3.4 Results

Proposition 3.1 *Let P be the synchronous product between S and TP (definition 3.1) and $\mathcal{T}(TP, S)$ be the DAG synthesized by applying the rules of definition 3.1. If $(q^{TP}_{init}, q^{S}_{init})$ is the root of the dag, then $q^{TP}_{init} \prec q^{S}_{init}$ else the test purpose and the specification are not consistent.*

Let $OT(S) = \{TP \in IOLTS \mid q^{\mathrm{TP}}_{\mathrm{init}} \prec q^{\mathrm{S}}_{\mathrm{init}}\}$ be the set of test purposes which are consistent with respect to the specification S. Let $TS(S) = \{\mathcal{T}(TP, S) \mid TP \in OT(S)\}$ i.e. the set of test cases (test suite) that can be constructed for a specification S. For $\mathcal{T} \in TS(S)$, we denote $Max\_traces(\mathcal{T}) = \{\sigma \mid \mathcal{A}(\mathcal{T} \textbf{ after } \sigma) = \emptyset\}$ the set of maximal traces of $\mathcal{T}$. For $\sigma = \sigma'.\alpha \in Max\_Traces(\mathcal{T})$, and for an implementation I, we define $\texttt{verdict}(\sigma, I) = \texttt{verdict}(\mathcal{T} \textbf{ after } \sigma', \alpha, 1)$. Notice that $\mathcal{T}$ is deterministic, thus $\mathcal{T}$ **after** σ' is unique. We have the two following results:

Proposition 3.2 *(Soundness) Assuming that timeouts are produced if and only if the implementation is deadlocked, for every implementation I, if the application of a test case $\mathcal{T} \in TS(S)$ produces a Fail verdict then I does not conform with S:*
$(\exists \mathcal{T} \in TS(S), \exists \sigma \in Max\_Traces(\mathcal{T}), \texttt{verdict}(\sigma, I) = Fail) \Rightarrow \neg(I \textbf{ ioconf} S)$.

This second proposition is not exactly the converse. Implementations can be non deterministic. Thus the application of the same sequence of actions of the tester may produce different verdicts. Thus, like other authors [13], we assume a bounded fairness hypothesis on implementations. This informally means that a bounded number of executions of a non deterministic implementation will show all its behaviours. For $n \in \mathbb{N}$, we define $\texttt{verdict}^*(n, \sigma, I)$ to be Fail if one of the n applications of σ on I produces a Fail verdict, Pass otherwise.

Proposition 3.3 *(Exhaustivity) For every implementation I, if I does not conform with S, there exists a test case $\mathcal{T} \in TS(S)$ which can produce a Fail verdict:*
$\neg(I \textbf{ ioconf } S) \Rightarrow (\exists \mathcal{T} \in TS(S), \exists \sigma \in Max\_Traces(\mathcal{T}), \exists n \in \mathbb{N}, verdict^*(n, \sigma, I) = Fail)$.

4 Experimentation

The algorithms and transformations described in previous sections have been developed in the CADP toolbox [6] as a software component named TGV (for Test Generation using Verification techniques). In order to prove the feasibility of the approach, we have applied TGV to an industrial protocol, the DREX protocol.

4.1 TGV

As we were primary interested by demonstrating the feasibility of our approach before a real implementation, all algorithms are not yet combined into a unique on the fly algorithm. We have used the Geode simulator [1]

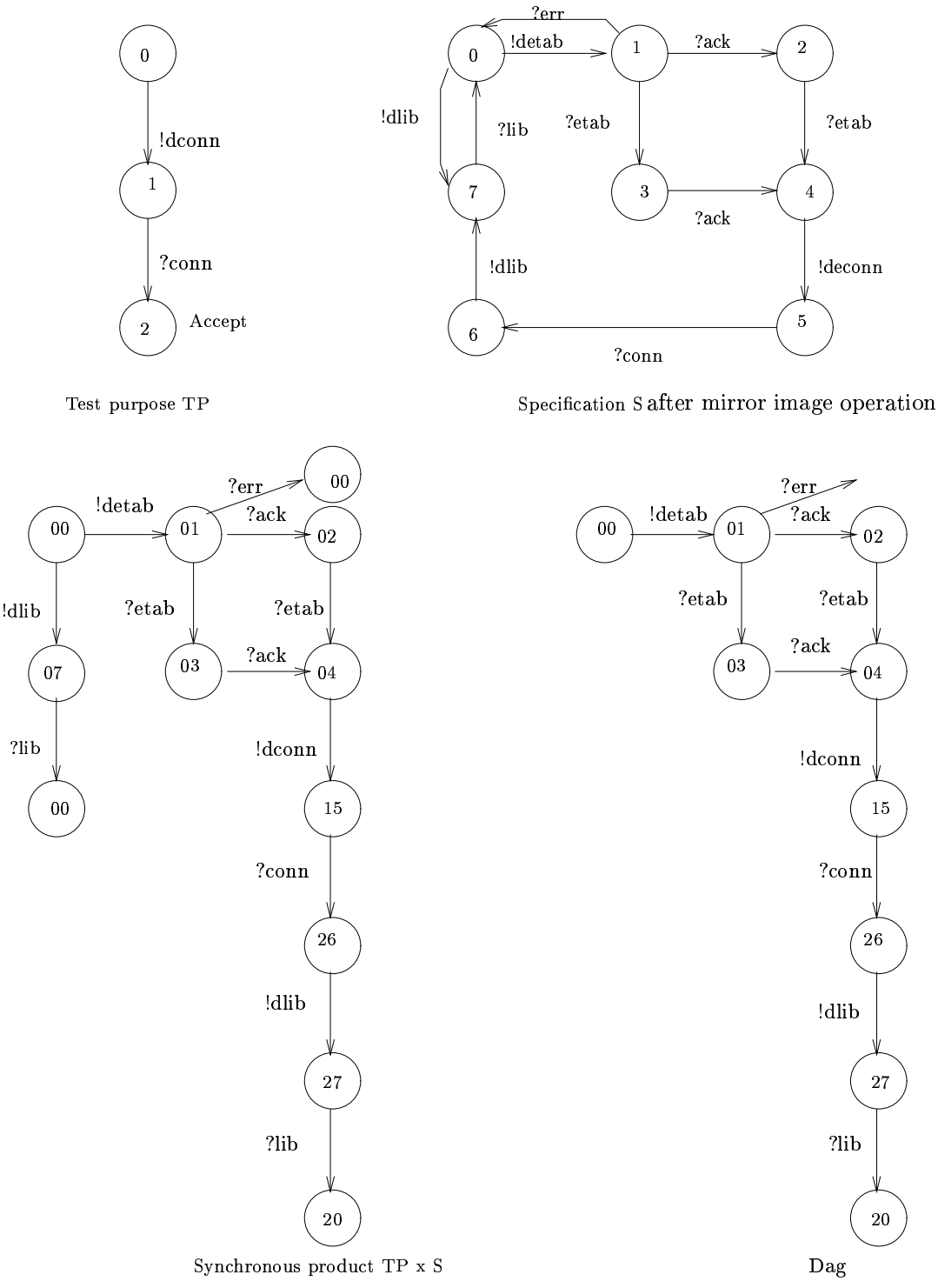

Figure 1: An example of synchronous product

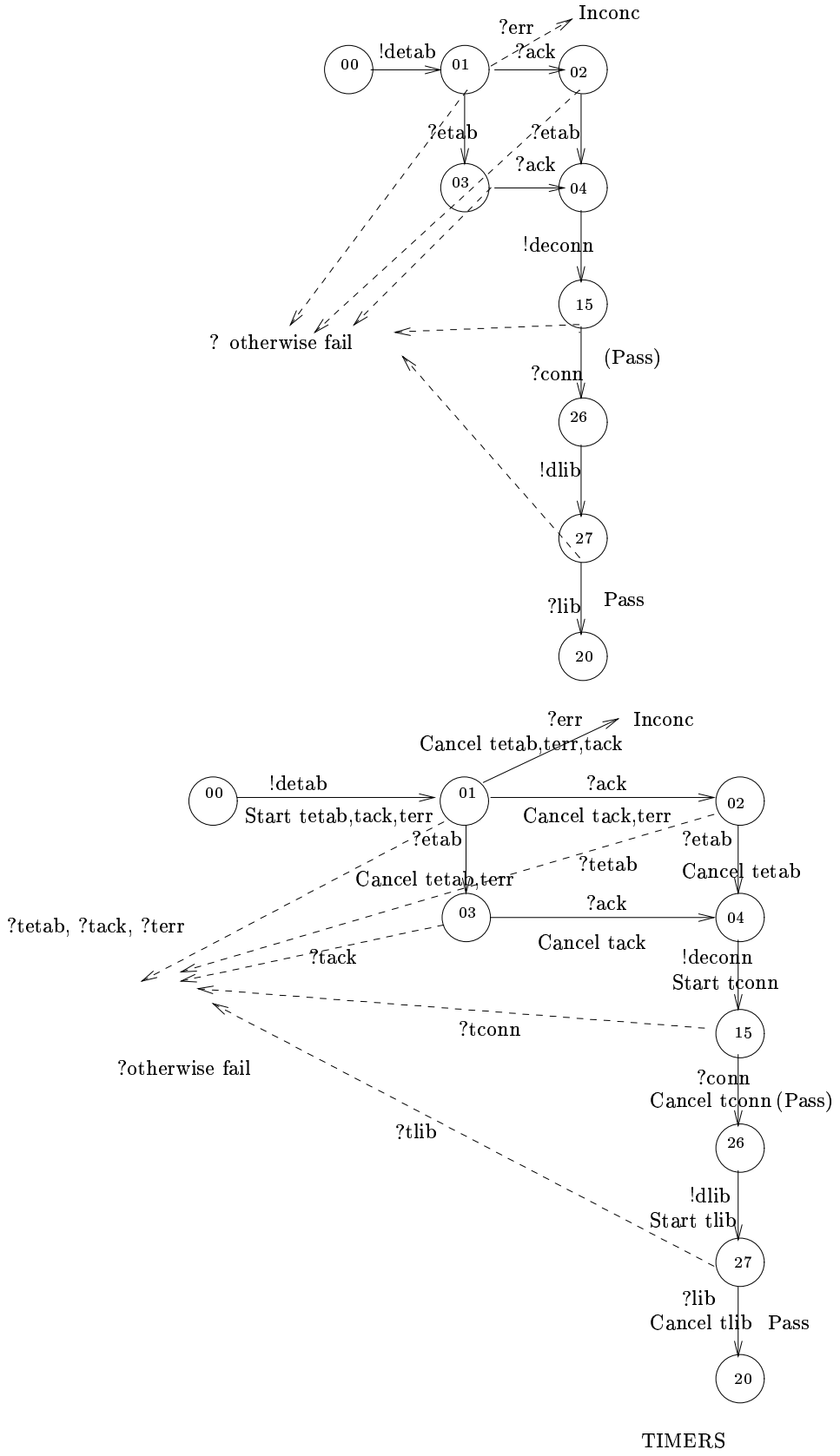

Figure 2: The generated test case

from Verilog as an SDL [3] front-end which produces state graphs representing the behaviour of a specification, constrained by the test purpose constraints.

Thus the inputs of TGV are a state graph produced by Geode (from a SDL specification of the protocol) and an automaton formalizing the behavioural part of a test purpose. The output is the behaviour description and constraints definitions of a test case in the standard TTCN format [10].

Different steps bring out this output. The first step takes as input the state graph produced by Geode and transforms it into a graph representing the observable behaviour of the protocol specification in the testing environment (*external view graph*). Several transformations are performed in this step: abstraction of unobservable internal actions, determinization, mirror image which transforms inputs into outputs and vice versa and construction of diamonds modelling concurrency introduced by the asynchronous interaction between the tester and the IUT. The next step is the kernel of TGV. The output is the DAG which contains all informations needed in TTCN test cases. The last step takes as input the DAG. The algorithm extracts from the transition labels the message parameters and produces the constraint part in TTCN GR format. The remaining graph is unfolded into a tree describing the behavioural part of the test case in TTCN GR format. Finally the constraint and behavioural parts of the test case are translated into the graphical format TTCN GR.

4.2 Experiment with the DREX protocol

TGV has been used during an industrial contract for the *Direction Générale pour l'Armement*. The protocol used for the experiment was a military protocol called the DREX protocol which allows the access to the transit network Socrate of the French Army, defined in the framework of Integrated Service Military Network. This protocol has been chosen for three main reasons: firstly, we wanted to prove the feasibility of automatic test generation methods on realistic specifications; secondly, an SDL specification of a similar protocol was already available, and finally, hand written test suites had already been produced. This last point is important as hand written test cases have served as a basis for comparison with automatically generated test suites.

The SDL specification models the behaviour of the DREX protocol on the network, communicating asynchronously with two users by two PCOs. The size of the SDL specification was about 2000 lines. 54 test purposes have been considered and 54 corresponding test cases have been generated. The time needed for the generation of a test case has to be separated into two parts: the time needed for the graph generation with Geode which took between 3.5s and 400s and the test case generation with TGV which took between 1s and 2s.

We have compared automatic test suites generated by TGV with hand written test suites in a qualitative way. Even though TGV is just a prototype, all hand written test suites or similar ones have been generated. The differences that were observed were principally due to the fact that TGV treats systematically concurrency and timers. For example, in some hand written test cases, concurrency between events were forgotten and risked an incorrect verdict. Some differences were also due to the formal interpretation of test purposes. More details and other quantitative results of this study can be read in [7].

5 Conclusion

In this paper, we have shown how on-the-fly verification techniques could be used in the generation of test suites. Starting from an already known conformance relation and from the experiment gained with the analysis of hand written test cases, we have formally defined the rules allowing a construction of complete test cases, with preambles, postambles, verdicts and timers. These rules allowed us to prove that generated test cases are sound (correct implementations are not rejected) and exhaustive (if we assume a fairness hypothesis on implementations under test, incorrect implementation can be detected) with respect to the conformance relation. A depth first search algorithm implementing these rules has been described. A first version of this algorithm has been implemented in a prototype named TGV which produces TTCN test suites from SDL specifications. TGV has been experimented on an industrial protocol, proving the efficiency and maturity of the algorithm.

The next step in this study will be the development of a new prototype which will incorporate the algorithm described in this paper in a unique on-the-fly algorithm and its integration in a complete validation tool. Another continuation of the work is to deal with concurrent testing and links with interoperability testing.

References

[1] B. Algayres, Y. Lejeune, F. Hugonnet, and F. Hantz. The AVALON project : A VALidatiON Environment For SDL/MSC Descriptions. In *6th SDL Forum, Darmstadt*, 1993.

[2] E. Brinksma. A theory for the derivation of tests. In S. Aggarval and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII, IFIP*, pages 63–74. Elsevier Science Publishers, B.V., North-Holland, 1988.

[3] CCITT/SGx/WP3-1, Specification and Description Language, SDL. *CCITT Recommendation Z.100*, 1988.

[4] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. Two approaches linking a test generation tool with verification techniques. In A. Cavalli and S. Budkowski, editors, *8th Int. Workshop on Protocols Test Systems, Evry, France*, pages 159–174, September 1995.

[5] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In E.M. Clarke and R.P. Kurshan, editors, *Computer Aided Verification, 2nd International Workshop, CAV'90, New Brunswick, NJ, USA*. Springer Verlag, LNCS 531, 1990.

[6] J.-C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A Tool Box for the Verification of Lotos Programs. In *14th International Conference on Software Engineering*, Melbourne, Australia, May 1992.

[7] J.-C. Fernandez, C. Jard, T. Jéron, and G. Viho. An experiment in automatic generation of test suites for protocoles with verification technology. under revision for SCP, 1996.

[8] J.-C. Fernandez and L. Mounier. On the fly verification of behavioral equivalences and preorders. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV'91, Aalborg, Denmark*, pages 181–190. Springer Verlag, LNCS 575, June 1991.

[9] J.-C. Fernandez, L. Mounier, C. Jard, and T. Jéron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1:251–273, 1992. Kluwer Academic Publishers.

[10] OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - part 2 : Abstract Test Suite Specification - part 3 : The Tree and Tabular Combined Notation (TTCN). *International Standard ISO/IEC 9646-1/2/3*, 1992.

[11] C. Jard and T. Jéron. On-line model-checking for finite linear temporal logic specifications. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France*, pages 275–285. Springer–Verlag, LNCS 407, June 1989.

[12] C. Jard and T. Jéron. Bounded memory algorithms for verification on the fly. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification, 3rd International Workshop, CAV'91, Aalborg, Denmark*, pages 192–202. Springer Verlag, LNCS 575, June 1991.

[13] M. Phalippou. *Relations d'implantations et Hypothèses de Test sur des automates à entrées et sorties.* Thèse de doctorat, Université de Bordeaux, France, 1994.

[14] M. Phalippou. Test sequence using Estelle or SDL structure information. In *FORTE'94*, Berne, October 1994.

[15] J. Tretmans. Testing Labelled Transition Systems with Inputs and Outputs. In A. Cavalli and S. Budkowski, editors, *8th Int. Workshop on Protocols Test Systems, Evry, France*, pages 461–476, September 1995.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399